PyTorch RPC: Distributed Deep Learning Built on Tensor-Optimized Remote Procedure Calls

Shen Li [1] Pritam Damania [1] Luca Wehrstedt [1] Rohan Varma [1] Omkar Salpekar [1] Pavel Belevich [1] Howard Huang [1] Yanli Zhao [1] Lucas Hosseini [1] Wanchao Liang [1] Hongyi Jia [1] Shihao Xu [1] Satendra Gera [1] Alisson Azzolini [1] Guoqiang Jerry Chen [1] Zachary DeVito [1] Chaoyang He [2] Amir Ziashahabi [2] Alban Desmaison [1] Edward Yang [1] Gregory Chanan [1] Brian Vaughan [1] Manoj Krishnan [1] Joe Spisak [1] Salman Avestimehr [2] Soumith Chintala [1]

Abstract

Distributed training technologies have advanced rapidly in the past few years and have unlocked unprecedented scalability with increasingly complex solutions. These technologies have made distributed training much more efficient and accessible, though they impose specific constraints on the training paradigm or the model structure. As a result, applications that fail to meet these constraints must rely on general-purpose distributed computing frameworks to scale out. However, without access to the internal components of deep learning frameworks, these distributed computing frameworks usually significantly fall short in terms of efficiency and usability. To address these problems, we propose PyTorch RPC as a generic and high-performance solution for distributed deep learning. Compared to generic distributed computing frameworks, PyTorch RPC natively provides essential features for implementing training applications in a distributed environment, including optimized tensor communications, remote memory management, and distributed autograd. Evaluations show that PyTorch RPC attains up to two orders of magnitude faster tensor communication compared to gRPC with one-tenth of the user code. Case studies further demonstrate that users can easily employ PyTorch RPC to build efficient reinforcement learning applications (video game solver), implement large language models (GPT3), train recommendation models (DLRM), and scale federated learning tasks (FedML).

1 Introduction

Innovations in distributed deep learning have powered unprecedented growth in data and model sizes. Natural language processing (NLP) and computer vision (CV) experts have recently explored multi-trillion parameter Transformer models (Brown et al., 2020; Fedus et al., 2021). To deal with such gigantic models, distributed training frameworks have advanced from vanilla data parallelism (Sergeev & Balso, 2018) and parameter servers (Li et al., 2014; Thangakrishnan et al., 2020) to pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019; Yang et al., 2021a) and operator-sharding-based parallelism (Chen et al., 2021; Jia et al., 2019; Rajbhandari et al., 2019; Shazeer et al., 2018; Shoeybi et al., 2019). Unfortunately, these distributed training solutions usually make assumptions on model structures and training paradigms, forcing developers to tailor algorithms to fit the existing tools. Applications that fail to meet these assumptions must build distributed training solutions from scratch, repeatedly reinventing lower-level components such as tensor-aware communication and distributed back-propagation. For example, libraries in reinforcement learning (Küttler et al., 2019), federated learning (He et al., 2020b), and graph learning (Lerer et al., 2019) have independently implemented communications on top of generic RPC frameworks, such as gRPC (gRPC, 2021), which are known to be inefficient for ML applications (Xue et al., 2019; Biswas et al., 2018) due to slow tensor transmissions and major feature gaps such as cross-process autograd. It is important to acknowledge that solutions with specific assumptions can usually explore more system optimization opportunities and certainly deserve investment, especially for widely adopted model structures and training paradigms. Nevertheless, rather than always dictating model training methodologies, we believe it is equally important to build generic distributed training tools and empower the community to innovate and thrive on new models and new domains.

This paper proposes PyTorch RPC, a framework with essential tools for generic distributed deep learning. Despite disparities across different training solutions, they all share

[1]Meta AI [2]University of Southern California. Correspondence to: Shen Li <shenli@meta.com>, Pritam Damania <pritam.damania@gmail.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

similar basic phases, namely, loss computation (`forward()`) and back-propagation (`backward()`). PyTorch RPC offers succinct programming interfaces to implement both phases in a distributed environment. For loss computation, applications can run arbitrary user functions remotely with highly efficient tensor communications. In addition, Remote Reference helps manage the life cycle of intermediate remote activations, hiding the complexity of distributed memory management. For back-propagation, Distributed Autograd stitches together cross-process autograd graphs and addresses graph discovery and termination detection. Therefore, as long as applications comply with similar loss computation and back-propagation phases, PyTorch RPC can serve as a powerful tool to help scale beyond machine boundaries. Compared to existing solutions that use third-party distributed computing libraries to scale deep learning applications, PyTorch RPC can uniquely take advantage of PyTorch non-user-facing components such as tensor storage allocations, CUDA stream pools, and the local autograd engine, and hence can deliver greater efficiencies.

We evaluated both the efficiency and the usability of PyTorch RPC. Results show that PyTorch RPC achieves up to 12x faster tensor communication than gRPC on the same communication media. When more communication channels (*e.g.*, InfiniBand) are available, the lead of PyTorch RPC reaches 100x. Moreover, in case studies, we used PyTorch RPC to implement a distributed video game solver, a 175B parameter GPT3-like (Brown et al., 2020) model, a DLRM (Naumov et al., 2019)-like model, and a cross-silo FedML (He et al., 2020b) task to verify the feasibility of supporting reinforcement learning, large natural language model training, recommendation system applications, and federated learning use cases.

2 Background and Challenges

With the rapid growth in data and model sizes, distributed training techniques such as data parallelism and model parallelism have naturally emerged. Data parallel training replicates the model on all processes, with each process consuming a different split of the input. Its straightforward single-program multi-data scheme has led to its widespread adoption across various applications. Many existing solutions have been consolidated to use collective communications (*e.g.*, `AllReduce`) to synchronize model replicas in every iteration (Li et al., 2020; Sergeev & Balso, 2018). When the device memory capacity falls short to host one full model replica, using model parallelism can help reduce per-device memory footprint by dividing the model into shards and placing them on multiple devices. Compared to data parallelism, model parallelism has evolved into multiple forms, including pipeline parallelism (He et al., 2021; Huang et al., 2019), sharded parameter servers (Li et al., 2014), and operator-sharding based parallelism (Rajbhandari et al., 2019). Although existing solutions have successfully helped scale some applications and domains, they apply rigorous constraints on model structures or training paradigms. For example, PyTorch Pipeline only accepts `nn.Sequential` as the input model, while `DistributedDataParallel` and ZeRO (Rajbhandari et al., 2019) require a single-program multi-data paradigm, where all processes need to run the same set of operations in the same order. Applications that fail to meet these constraints have to resort to generic distributed computing tools such as gRPC to glue together the training algorithm. However, without access to PyTorch internal components such as tensor allocation, CUDA stream pool, and autograd engine, generic distributed libraries can hardly deliver competitive training efficiency.

PyTorch RPC aims to offer both high usability and high efficiency for a wide spectrum of ML applications and training paradigms. Challenges are four-fold.

- **Programming Interface** has to maintain a balance between frontend usability and backend flexibility such that ML practitioners can easily adopt it while platform developers can introduce system optimizations.
- **Tensor Communication** efficiency serves as the bedrock for the entire system, which needs to minimize the number of copies and maximize the overlap between communication and computation.
- **Memory Management** must free obsolete tensors as soon as possible in a distributed environment, given that device memory space is usually a scarce resource.
- **Distributed Autograd** must extend the local autograd engine beyond process boundaries to discover graph and detect termination in a distributed manner.

PyTorch RPC is carefully designed to address these challenges to simplify and scale distributed training.

3 Design

As a generic distributed training library, PyTorch RPC exposes APIs to directly scale fundamental training phases, namely, `forward()`, `backward()`, and `optimizer.step()`. To speed up tensor transmissions, the communication stack introduces multiple optimizations to overlap computation, tensor allocation, device-host communication, and cross-machine communication. For distributed memory management, PyTorch RPC offers a Remote Reference tool to track remote objects and automatically handle the object lifetime with negligible overhead. Finally, Distributed Autograd seamlessly stitches together local autograd engines across processes and machines.

```
1  from torch.distributed import autograd, optim, rpc
2
3  # Prepare model sub-layer
4  class Layer(torch.nn.Module):
5    def __init__(self):
6      super().__init__()
7      self.net = nn.Sequential(
8        nn.Linear(20, 20),
9        nn.ReLU()
10     )
11
12   def forward(self, rx):
13     return self.net(rx.to_here())
14
15 # Assemble model
16 class Model(torch.nn.Module):
17   def __init__(self):
18     super().__init__()
19     self.rl1 = rpc.remote("p1", Layer)
20     self.rl2 = rpc.remote("p2", Layer)
21   def forward(self, ri):
22     rx = rpc.remote(self.rl1.owner(),
           Layer.forward, args=(ri,))
23     ry = rpc.remote(self.rl2.owner(),
           Layer.forward, args=(rx,))
24     return ry
25
26 # Code below run on p0
27 m = Model()
28 opt = optim.DistributedOptimizer(
29   torch.option.SGD,
30   param_rrefs(m)  # Collect RRefs of parameters.
31 )
32
33 for batch in next_batch():
34   with autograd.context() as ctx:
35     ri = rpc.RRef(batch)
36     loss = m(ri).to_here().sum()
37     autograd.backward(ctx, [loss])
38     opt.step(ctx)
```

Listing 1: A Simple Model Parallel Example

3.1 Programming Interface

The goal of PyTorch RPC is to provide a programming model for machine learning system or middleware developers to easily create high performance solutions, such as parameter server paradigms, reinforcement learning systems, pipeline parallelism etc. The `RPC` package contains a rich set of features. For simplicity, this section uses the minimum number of APIs to demonstrate the concepts of Remote Execution, Remote Reference, and Distributed Autograd. The code snippet in Listing 1 uses a simple model parallel example to showcase the usage. It first defines a `Layer Sequential` module that contains a `Linear` and a `ReLU` as sub-modules. Its `forward()` function takes a Remote Reference (`RRef`) of the input and calls `to_here()` to fetch the data referenced by the `RRef`. Next, it defines the `Model` to assemble two `Layer` instances that are placed on processes `p1` and `p2` respectively. The `rpc.remote()` function leaves the result object on the callee and immediately returns an `RRef` pointing to the result. `Model`'s `forward()` function also takes an `RRef` of the input and passes it to the first `Layer` module. Line 23 uses Remote Execution to run the `forward()` function of the first `Layer` module on its owner process `p1` and returns an `RRef` of the intermediate output. After that, the returned `RRef rx` is passed to `rl2`. In this way, the intermediate output can be directly transmitted from `p1` to `p2` without going through the coordinator process `p0` where the `Model` instance resides. Finally, the forward and the backward passes are similar in local training, except that they live in a distributed autograd context which uses exclusive storage to hold gradients. This example shows how the `RPC` package simplifies the implementation of distributed training. We skip `DistributedOptimizer` in this paper as it is simple helper implemented using Remote Execution and Remote Reference. Next, we will dive into designs under the hood and revisit this example repeatedly to explain all details.

3.2 Tensor-Aware Communication

Compared to generic distributed computing systems (gRPC, 2021; Ray, 2023; Dask, 2023), PyTorch `RPC` *understands tensors* and comes with advanced tensor communication optimizations. As `Tensors` can be significantly larger than the rest of the message, they require extra consideration. When the caller serializes a request, PyTorch `RPC` extracts `Tensor` objects and keeps them intact. Non-`Tensor` items will be packed into a binary payload and sent over the most performant CPU channel (*e.g.*, TCP, SHM, etc.). This design allows PyTorch RPC to conduct two sets of optimizations, 1) leverage out-of-band fabrics (NVLink, IB, etc.) for accelerated `Tensor` transfers, and 2) pipeline and overlap multiple computation and communication stages.

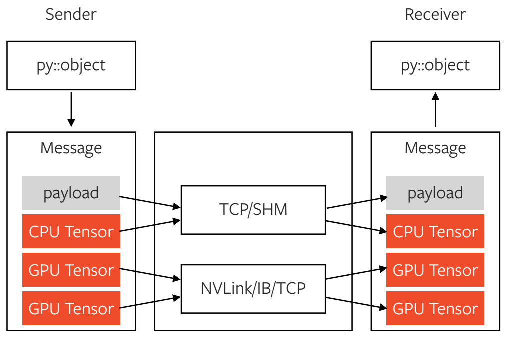


Figure 1: Tensor-Aware Communication

PyTorch RPC supports multiple types of channels and can automatically detect the optimal channel between the sender and receiver based on the storage device type and network interface availability. Figure 1 depicts the high-level idea. For each pair of sender and receiver, the sender passes the binary payload and a list of `Tensor` objects to the communication layer. Then, based on the device type of each `Tensor`, the communication layer detects the optimal channel by matching the available channels on both ends and picks the one with the highest priority (e.g., RDMA, SHM, or TCP).

After exchanging the meta information, the receiver learns the size and the device type of the `Tensor`, allocates `Tensor` storage accordingly, and transmits `Tensor` values through the chosen channel. Since it is closely coupled with PyTorch internal components, the communication stack can avoid creating unnecessary `Tensor` copies and use the optimal communication channel for each individual `Tensor`.

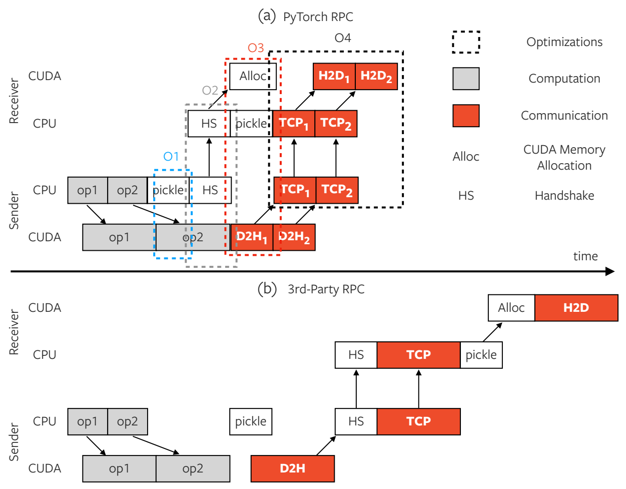


Figure 2: Comparison with Third-Party RPC

Decoupling the payload and `Tensor` objects also further exposes opportunities to overlap CPU computations, CUDA computations, CUDA memory allocations, TCP communications, and host-device communications, leading to highly efficient CUDA `RPC`. Figure 2 (a) uses an example to illustrate the performance optimizations, assuming the `Tensor` objects travel through TCP instead of RDMA channels. The example contains two CUDA operations followed by a single `Tensor` communication. Each solid block represents a computation or communication, and each dashed block denotes one type of optimization. First (`O1`), since the pickling algorithm no longer needs to read `Tensor` values, it can take advantage of the asynchronous behavior of CUDA computations and start without waiting for pending CUDA operations, which overlaps CPU computations with CUDA computations, as shown by the blue dashed block. Second (`O2`), the handshake and the payload communication can also run concurrently with CUDA computations as they only require the shape and the device information of the `Tensor` which are available as soon as the CPU thread appends the CUDA operation to the stream. Third (`O3`), upon receiving the shape and the device information, the receiver can allocate `Tensors` on the destination device, further overlapping memory allocation with sender CUDA computation and communication. Finally (`O4`), the communication layer of PyTorch RPC uses persistent pinned staging buffers to enable non-blocking device-to-host (D2H) and host-to-device (H2D) communications. With this design, tensor data is copied into the staging buffer in chunks and can start sending the first chunk over TCP while running the D2H copy on the second chunk concurrently. This enables overlapping D2H communication with TCP communication. In contrast, Figure 2 (b) demonstrates the behavior when using a generic distributed computing library. The pickling step has to wait for all pending CUDA operations since it needs to read `Tensor` values. Additionally, the handshake is blocked by the D2H copy in the pickling step because it must know the pickled message size. Without understanding CUDA tensors, the user application has to explicitly move `Tensor` objects from CPU to CUDA, forfeiting the opportunity to overlap TCP communication with H2D communication. As a result, these large bubbles in the critical path force generic distributed computing frameworks to suffer from longer communication delays.

3.3 Memory Management

Deep learning applications require tracking the lifetime of activations to make sure they are garbage collected neither too early before usage in the `forward()` and the `backward()` passes nor too late to avoid out-of-memory errors. Local training can easily meet these requirements as all contexts stay within a single process and regular reference counting and garbage collection should suffice. In a distributed environment, global context becomes a luxury. However, the requirements on tracking lifetime, referencing remote data, and managing memory spaces remain the same. PyTorch RPC addresses the problem by introducing Remote Reference with a dedicated protocol to conduct distributed reference counting and garbage collection. Each `RRef` has a single owner and an arbitrary number of users, forming a star topology. The owner lives on the same process holding the data of the `RRef` and performs bookkeeping for all users. When sharing an `RRef` across processes, the RPC framework automatically generates control messages to update the reference count on the owner. The `RRef` reference count protocol can handle all $\{owner, user\} \times \{caller, callee\}$ combinations. In general, the sender that shares an `RRef` is responsible for keeping its local `RRef` alive until the receiver confirms the readiness of the new `RRef` instance, and the receiver with the new `RRef` is responsible for notifying the `RRef` owner. When the reference count drops to zero, the owner will delete the `RRef`.

The example in Listing 1 created multiple `RRef` instances. Figure 3 shows the ownership graph. Process `p0` serves as a coordinator in this example and runs the training loop. It holds user `RRefs` to the two sub-modules (*i.e.*, `r11` and `r12`) and grabs `RRefs` of inputs and outputs (*i.e.*, `ri`, `rx`, and `ry`) to drive the forward pass. The data of the intermediate output `rx` is directly transmitted to `p2` from `p1`, while `p0` can stay out of that heavy-weight data path.

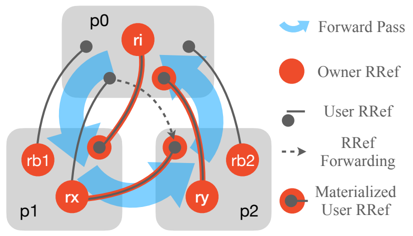


Figure 3: RRef Ownerships

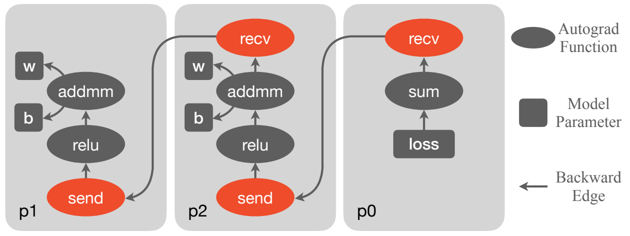


Figure 4: Distributed Autograd Graph

Creating and sharing `RRef` instances heavily rely on communications. Therefore, apart from lifetime tracking and referencing remote objects, `RRef` also needs to minimize the additional delay introduced to the forward and the backward passes. To address this problem, `RRef` is designed to be immediately available for using and sharing after creation (*i.e.*, `rpc.remote()`) returns without waiting for any control messages, while data access (*i.e.*, `to_here()`) always blocks till the completion of preceding computations that fill `RRef` value. This design excludes the `RRef` control communication delay from the main computation path, and can overlap with pending CUDA computations.

3.4 Distributed Autograd

The existing autograd engine in PyTorch only works within the context of a single process, and it requires global information to conduct graph discovery and termination detection. In the distributed environment, global knowledge becomes a luxury. However, PyTorch RPC should hold the same usability standard as local training. PyTorch RPC offers a similar `.backward()` API to launch automatic differentiation on a distributed autograd graph. In order to propagate the backward pass to all participating processes and remote autograd graphs, the distributed autograd engine inserts a pair of `send` and `recv` autograd functions for each `Tensor` communication during the forward pass and tracks them in a dedicated distributed context. The backward pass then uses those `send` and `recv` functions to stitch together the distributed autograd graph.

The distributed autograd graph of Listing 1 is depicted in Figure 4. The first `Layer` module (*i.e.*, `rl1`) lives in `p1` and contains an `addmm` function and a `relu` function. In the forward pass, the intermediate output from `rl1` is fed to `rl2` as the input. Therefore, there is a pair of `send` and `recv` functions connecting local autograd graphs on `p1` and `p2` when calling `to_here()`. The autograd edge in between points from `recv` to `send` denotes the direction of gradient communications during the backward pass. The second pair of `send` and `recv` functions connect `p2` and `p0`, which are installed when `p0` calls `to_here()` on `Model`'s output `RRef`. With this distributed autograd graph, the backward pass can automatically reach out to `addmm` functions and calculate gradients for all participating parameters.

Under the hood, PyTorch RPC introduces a layer on top of the local autograd engine to handle incoming and outgoing communications and conducts local autograd graph discovery. Compared to the local `.backward()` where the root node is usually the loss tensor and leaf nodes are usually `AccumulateGrad` functions, the distributed autograd engine must recognize local `send` autograd functions as additional roots and local `recv` functions as additional leaves. To propagate autograd to remote participating processes, each `recv` autograd function sends an RPC message to the process hosting its peer `send` autograd function, which triggers the same graph discovery and autograd propagation algorithm on that process. The distributed autograd engine recursively applies this procedure until reaching all involved processes. To remain consistent with the local autograd `.backward()` function, the distributed autograd engine also recursively waits for all remote distributed autograd executions triggered by its local `recv` functions. In this way, the `.backward()` invocation on the root process only returns when gradient `Tensors` are ready on all participating processes. However, due to the asynchronous behavior of CUDA RPC, some local or remote CUDA operations producing the gradients might still wait in the stream, meaning that gradient `Tensor` values cannot be consumed yet. Hence, the distributed autograd engine applies proper synchronization by recording a CUDA event after accumulating gradients and uses that event to block all read operations on the gradient `Tensors`.

Compared to the local `.backward()` function, which always stores the gradients in the `.grad Tensor`, Distributed autograd uses a per-iteration context to wrap all `send` / `recv` autograd functions created in the forward pass and gradients generated in the backward pass. There are two main reasons for this design decision. First, unlike local training, no single process has a global picture of the distributed autograd graph and therefore cannot independently detect the completion of the autograd and parameter updates to trigger garbage collection. It is possible to handle this problem by carrying additional information in the forward pass or relying on additional communications to detect the distributed autograd graph before launching the backward pass. However, it would inevitably introduce additional overhead to the already mission-critical backward pass. Having a context allows for more efficient and straightforward cleanups. Second, in distributed training, there can be multiple concurrent backward passes accessing the same parameters (Recht et al., 2011). Each backward pass requires a dedicated context to store gradients to avoid read and write conflicts before the gradients are consumed. Applications can create a context using `autograd.context()` as shown on line 36 in Listing 1 and pass the context to the backward pass on line 39. The context contains an identifier used by all participating processes to look up gradients and autograd functions. For each individual Distributed Autograd Graph, the backward pass needs to wait for every process involved in the computation to complete the backward pass. As a result, any stragglers could slow down the entire backward pass. However, this issue can be mitigated by leveraging asynchronous training paradigms (*e.g.*, with parameter servers) which have multiple concurrent and independent Distributed Autograd Graphs.

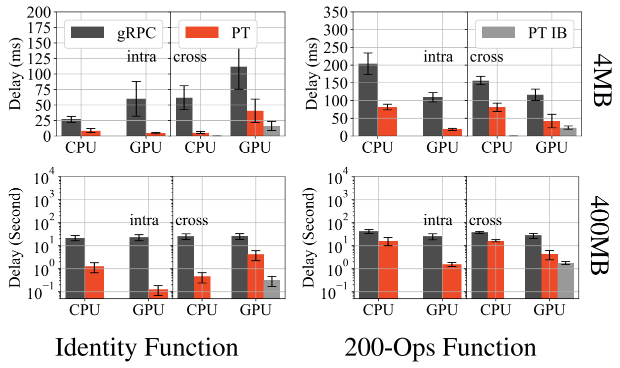


Figure 5: RPC Latency: All figures share the same legends. PT IB means PyTorch RPC over InfiniBand. The texts `intra` and `cross` refer to intra-machine and cross-machine.

4 Evaluation

This section demonstrates how PyTorch `RPC` can help facilitate efficient distributed training. Section 4.1 measures individual RPC latency on various hardware interconnects. Section 4.3, 4.4, and 4.5 present three case studies on reinforcement learning, large language model training, and recommendation systems. Experiments focus on training speeds of different implementations, including:

- **gRPC** uses PyTorch for computations and relies on gRPC (gRPC, 2021) for communications. The gRPC framework is a popular solution for distributed computing which has been used in conjunction with PyTorch in various machine learning applications (He et al., 2020b; Küttler et al., 2019).
- **CPU** uses PyTorch RPC for communications, but only CPU RPC channels are enabled.
- **CUDA** employs full-fledged PyTorch where GPU-direct communication is enabled.

Individual case studies compare a slightly different set of experiments, which will be explained subsequently. Please note that these experiments aim to show the flexibility and efficiency of PyTorch RPC rather than developing the best models or most performant training paradigms.

Experiments are run on two different clusters. A small in-house cluster contains two servers connected by InfiniBand, where each server has 8 16GB Tesla V100 GPUs. A larger AWS cluster contains 15 servers connected by `4X100Gb` Ethernet, where each server has 8 32GB Tesla V100 GPUs.

4.1 RPC Latency

To measure the raw RPC latency, each experiment makes 10 asynchronous RPC from the same caller to the same callee. The RPC function takes one `Tensor` and returns one `Tensor` of the same size. Figure 5 shows the results. The eight figures are grouped into 4 pairs, and the 4 pairs are organized into 2 rows and 2 columns. Within the same row, all experiments use the same `Tensor` size, which is marked on the right. Within the same column, all experiments use the same user function. The `Identity` function returns the argument `Tensor` as-is. The `200-Ops` function runs 200 point-wise arithmetic `Tensor` operations on the argument `Tensor` and returns the result. Each pair of figures conducts the same experiments on one machine (intra) and across two machines (cross) respectively. In cross-machine GPU Tensor experiments, we measure two sub-scenarios where PyTorch RPC uses `4X100Gb` Ethernet and InfiniBand respectively. InfiniBand experiments are marked as `PT IB`. Each individual figure compares CPU and GPU `Tensors`. PyTorch RPC outperforms gRPC in all scenarios. The largest gap reaches two orders of magnitude which occurs in cross-machine experiments that use 400MB GPU `Tensor` and `Identity` function. The speedups came from multiple optimizations. Firstly, PyTorch RPC has been intricately incorporated with PyTorch internals to prevent redundant copying, whereas type conversions are inevitable when passing `Tensor` objects from PyTorch to gRPC. Second, as illustrated in Figure 2, PyTorch RPC aggressively overlaps communication with computation. In contrast, gRPC mandates stream synchronization before pickling and data transmission, thereby relinquishing the prospect of overlapping. Furthermore, PyTorch RPC employs diverse hardware-related optimizations when feasible, such as pinned memory staging buffers, cross-memory attachment, NVLink, etc.

Beyond the experiments, using PyTorch RPC can implement communication with less than 10 LoC by simply calling `init_rpc`, `rpc_async`, and `shutdown`. In contrast, when using gRPC, we must specify the message structure in the `.proto` file, build it, and implement the client and the server separately, which takes around 100 LoC and will become increasingly complicated for more advanced use cases. One interesting observation is that switching to more expensive user functions leads to a larger regression in delay for gRPC than in PyTorch RPC, although the amount of computation is the same for both experiments. We believe this is because when using gRPC the application needs to convert `Tensors` into gRPC messages, which elongates the Global Interpreter Lock (GIL) occupation time on each thread. When combined with more expensive user functions, it would result in worse GIL contentions and longer GIL queuing delays.

4.2 Control Overhead

Experiments in this section measure the control overhead of Remote Reference and Distributed Autograd and quantify its impact to the training critical path.

Tensor Size	CPU Tensor	CPU RRef	CUDA Tensor	CUDA RRef
4MB	4.4±0.6ms	2.8±0.7ms	0.5±0.07ms	0.8±0.1ms
40MB	52±5ms	46±1.3ms	2.0±0.07ms	2.3±0.05ms
400MB	513±19ms	541±31ms	18.6±0.07ms	19.1±0.2ms

Table 1: Remote Reference Overhead

We first measure the overhead of Remote Reference by passing an identity function to `rpc_sync`, where the function either takes a `Tensor` and returns it as is or takes an `RRef` of a `Tensor` and returns `rref.to_here()`. Results are shown in Table 1, where each experiment is repeated 100 times across two processes on the same machine. CPU communications suffer from large noise that overshadows the overhead of Remote Reference. We believe this is because PyTorch does not cache storage for CPU `Tensors`, which contributes to a considerable portion of the total delay. CUDA experiments benefit from a much smaller noise after warm up and demonstrate that using `RRef` incurs around a 0.3ms-0.5ms delay, which meets expectations as it pays an additional round-trip.

norpc-fwd	nograd-fwd	grad-fwd	norpc-bwd	grad-bwd
1.2±0.08ms	1.6±0.12ms	2.0±0.10ms	0.7±0.09ms	1.6±0.13ms

Table 2: Distributed Autograd Overhead

To isolate the delay introduced by Distributed Autograd, we can measure the end-to-end forward and backward delays and then subtract the time spent on RPC, RRef, and local autograd. We use a simple model with two `1k x 1k` CUDA `Linear` layers in the experiments. Table 2 shows the results. We first measure the forward pass latency with (`nograd-fwd`) and without (`norpc-fwd`) RPC and RRef. The difference between these two is the RPC and RRef delay of the forward pass activation communication, which should be close to the backward pass gradient communication (1.6ms - 1.2ms = 0.4ms). We then measure the forward (`grad-fwd`) and backward (`grad-bwd`) latency when enabling Distributed Autograd and the backward latency without RPC (`norpc-bwd`). With these numbers, we can infer the Distributed Autograd forward overhead as `grad-fwd` - `nograd-fwd` = 2.0ms - 1.6ms = 0.4ms, and the backward overhead as `grad-bwd` - `norpc-bwd` - inferred RPC&RRef overhead = 1.6ms - 0.7ms - 0.4ms = 0.5ms.

In real applications, RPC target functions are usually much more expensive than a fraction of a millisecond. Moreover, as described in Section 3.2, these additional control communication overhead can overlap with pending CUDA operations and avoid the critical path, making the control overhead negligible

4.3 Scaling Reinforcement Learning Applications

	gRPC	CPU	CUDA
LoC Comm	52 (+270 gen)	10	12
LoC Comp	258	258	258
Rewards Max=700	654.3 ± 104.0	676.3 ± 52.03	674.0 ± 83.83

Table 3: Line of Code and Rewards

This case study shows how RPC and RRef assist in scaling out reinforcement learning tasks performed on an OpenAI Gym (Gym, 2021) environment. The application implements a well-known policy-based algorithm called the Asynchronous Advantage Actor Critic (A3C) (Mnih et al., 2016). In A3C, the worker agents are trained independently and update a global network that holds shared parameters. Agents each have their own network parameters and a copy of the environment. The agent independently interacts with a local environment and performs the loss calculation and backward pass. Rather than updating its own weights, each agent applies the gradients to a global network and then pulls the global network for the next training episode. More specifically, each training process independently performs the following four steps: 1) fetches the global network; 2) uses the latest network to interact with the environment and collect observations; 3) calculates losses with the observations and computes gradients; 4) applies gradients to the global network. We use gRPC and PyTorch RPC to carry out communications in steps 1 and 4 and compare the latency. Figure 6 shows the communication delays for fetching and

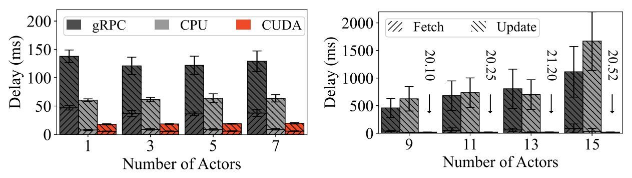


Figure 6: Video Game Solver Communication Delay

updating models. When there are at most 7 actors, the 8 processes (including the model parameter server) all reside on the same server where each process exclusively occupies a GPU. With more than 7 actors, processes are split across two different servers. CPU RPC outperforms gRPC when all processes reside on the same machine. However, when the actors span two machines, both gRPC and CPU RPC perform poorly and suffer from significant slowdowns. CUDA RPC, on the other hand, maintains a stable delay at around 20ms which is over 50x faster than gRPC with 15 actors.

4.4 Training Large Language Models

This section demonstrates the feasibility of training large language models using RPC. Experiments in this section skip gRPC since numbers presented in Section 4.1 and Section 4.3 already confirm that PyTorch RPC significantly outperforms gRPC in efficiency and simplicity for `Tensor` communications. Moreover, distributed autograd is much more complicated than sheer communication. Even if we build gRPC-based distributed autograd, it is still difficult to justify whether the implementation is equally optimized compared to the PyTorch distributed autograd engine. Therefore, this section focuses more on the efficiency of PyTorch RPC and RPC-based pipeline parallel on 1.3B- and 175B-parameter language models.

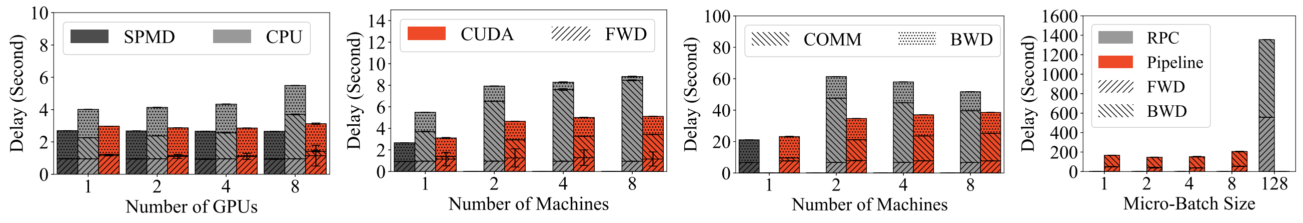
The first set of experiments employ a 1.3B parameter BERT model (Devlin et al., 2019) with WikiText (Merity et al., 2016) dataset to compare PyTorch RPC against single-process multi-device model parallelism (SPMD). The SPMD solution places model shards on different GPUs and relies on `Tensor.to(device)` to move activations to target devices. Under the hood, `Tensor.to(device)` uses the optimal channel (NVLink or PCIe) for device-to-device communications. Since all states live in the same process, SPMD represents the optimal speed that RPC can achieve without pipeline parallelism. In PyTorch RPC experiments, we spawn one process exclusively for each GPU. Cross-GPU communications always travel through RPC. Figure 7 (a)-(c) show the results. In Figure 7 (a), all experiments are conducted on the same machine (*i.e.*, communications for SPMD and CUDA RPC use NVLink if available). CPU RPC is significantly slower, which is expected as it introduces an additional `D2H` (Device-to-Host) copy on the caller, misses the faster NVLink channel, and introduces another `H2D` copy on the callee. CUDA RPC attains similar communication delay as SPMD with 1, 2, and 4 GPUs, but the delay jumps nearly 10x when using 8 GPUs, which is caused by absence of an NVLink between the 4th and 5th GPU. SPMD suffers less, as it only requires one `D2H` and one `H2D` copy within the same process, while CUDA RPC has to launch an additional `H2H` communication across the sender and the receiver processes. Transferring data across RPC process boundaries inevitably introduces higher overhead, but the benefit is that it does not require GIL and can easily scale to multiple machines to support larger models and different training paradigms. In Figure 7 (b) and (c) all experiments employ 8 GPUs. For an `n`-machine experiment, each machine contributes its first `8/n` GPUs. In these experiments, cross-machine communication delay becomes more dominant, especially as this cluster is not equipped with InfiniBand. There is no CPU bar in Figure 7 (c) on one machine because it hits the out-of-memory (OOM) error.

We then evaluate the feasibility of training a 175B-parameter model with RPC using a GPT3-like (Brown et al., 2020) architecture and show the impact of enabling pipeline parallelism. Naive model parallelism (RPC or SPMD) suffers from low device utilization, since only one device is busy at any given time instance. With pipeline parallelism, each mini-batch is split into multiple micro-batches which are then fed into the model in a pipeline (Huang et al., 2019; Kim et al., 2020). The backward path is handled by distributed autograd. Figure 7 (d) shows the result where the batch global size equals 128. Compared to the naive RPC solution, pipeline parallelism significantly speeds up training by 9x. The optimal speed is 146.3 Seconds per iteration when the micro-batch size is set to 4.

4.5 Supporting Recommendation Models

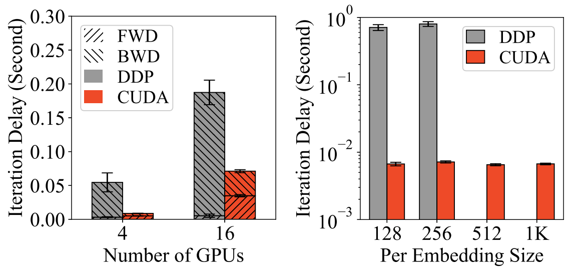
Recommendation models (Gupta et al., 2019; Naumov et al., 2019; Zha et al., 2023; Zhang et al., 2022) usually mix dense and sparse parameters. To train such models in a distributed environment, a naive solution is to wrap the entire model with `DistributedDataParallel` (DDP) and use collective communications to synchronize both dense and sparse gradients. However, replicating the entire model on every process does not scale to large embedding tables. Moreover, the collectives are known to be slow for sparse tensors, as they must perform one communication to gather the number of non-zero elements on all peers, then pad and allocate a buffer, and finally launch another communication to gather sparse tensor indices and values. A more efficient solution would put the embedding table on (sharded) parameter servers and rely on DDP to synchronize dense layers and RPC to update sparse layers.

In this section, we implement a DLRM-like model that consists of one embedding table and multiple MLP layers that contain 5 `Linear` (sizes: 2000, 1024, 1024, 512, 256, 64) with `ReLU` layers in between and one `Sigmoid` at the last layer. The number of embeddings for all experiments is 20 million. The DDP solution replicates the entire model on all processes and uses NCCL for dense gradients and Gloo for sparse gradients (NCCL does not support sparse tensors). The RPC solution still wraps the dense layers with DDP, but the embedding table with sparse features resides on a remote parameter server. Results are presented in Figure 8. Experiments skip CPU RPC, as previous sections have already confirmed that CUDA RPC significantly outperforms



(a) Single-Node, BS=32 (b) Multi-Node, BS=32 (c) Multi-Node, BS=256 (d) 175B Pipeline

Figure 7: Language Model Training Iteration Delay: Figures (a)-(c) share legends. Computation always resides on GPU. Communications are handled differently with 1) Single-Process Multi-Device model parallel (**SPMD**), 2) Data travel through host memory (**CPU**), 3) GPU-to-GPU direct (**CUDA**), 4) RPC-based pipeline parallelism (**Pipeline**). Figure (d) uses the same global batch size of 128 for RPC and pipeline parallelism, however for pipeline parallelism specifically we plot multiple results for micro-batch sizes 1, 2, 4 and 8.



(a) Dim = 64 (b) Varying Dim

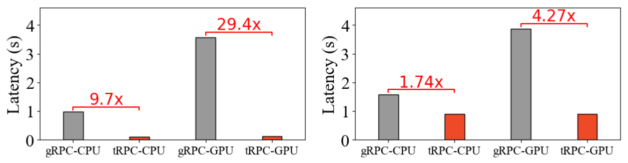
Figure 8: Recommendation Model Delays

CPU RPC. Figure 8(a) shows that with sparse gradients CUDA RPC beats collective communication-based DDP with a large lead. The forward pass of CUDA RPC is slower because it needs to perform a remote lookup, while DDP's embedding table is local. Figure 8(b) evaluates the limit of the embedding size of the two solutions. DDP hits the OOM error when per-element embedding size reaches 512, while RPC-based solution can scale to larger embedding tables.

4.6 Strengthening Federated Learning Framework

PyTorch RPC is also particularly suitable for improving the flexibility and communication performance of federated learning (FL) (Kairouz et al., 2021; Wang et al., 2021), a disruptive and promising distributed learning paradigm that aims to protect user privacy in a decentralized training manner. Compared to the three use cases mentioned above, FL typically has more complex communication patterns, and the communication bottleneck is more prominent because FL involves edge servers (e.g., servers belonging to a medical institute or a bank) that do not have the high bandwidth network adapter, such as InfiniBand or EFA.

We have successfully integrated PyTorch RPC into FedML (He et al., 2020b), a PyTorch-based federated learning framework widely used in academia. Using PyTorch RPC-enabled FedML, we evaluate the communication performance for a real use case in a cross-silo cross-account FL where the private datasets of FL clients are located at multiple AWS accounts but share the same AWS region. In this setting, we use AWS EC2 p3.2xlarge (8 CPU Cores, 1 Tesla V100 GPU) as the edge server (FL client) and configure two bandwidths (10Gb/s and 1Gb/s) to better understand the performance in different real use cases. We record the communication latency of gRPC and PyTorch RPC (tRPC) when running conventional FL algorithms (FedAvg and FedSGD) to train a model with 25M parameters on both a CPU Tensor and a CUDA Tensor. The results, shown in Figure 9, demonstrate that tRPC can achieve significant improvement even in the distributed training scenarios where GPU clusters do not have access to high bandwidth interconnects (e.g., AWS EFA that has 400Gb/s bandwidth or InfiniBand that normally has a bandwidth larger than 100Gb/s). Especially, tRPC can reduce the latency time from a few seconds (> 3) in gRPC to less than 1 second. This improvement can accelerate the training speed to a large degree because the training typically requires many communication rounds (Konečný et al., 2016). tRPC even provides more end-to-end training time acceleration when researchers prefer FedSGD, which synchronizes the weights after only one mini-batch for faster convergence, rather than FedAvg, which does weight aggregation after one local epoch.



(a) Bandwidth (10Gb/s) (b) Bandwidth (1Gb/s)

Figure 9: Federated Learning Experimental Results

PyTorch RPC can also offer simplicity for diverse algorithmic scenarios in FL. There are especially three algorithmic use cases in FedML that involves complex communication protocols but their implementations have been substantially simplified: LightSecAgg (Yang et al., 2021b) shows the case that the federated averaging algorithm (Konečnỳ et al., 2016) is empowered by secure aggregation, which requires multiple rounds of transmitting high frequent security protocol-related messages in between the training loop; vertical federated learning (Hardy et al., 2017; Yang et al., 2019) enables the collaborative learning when the entire feature space is isolated to multiple organizations so that complex security and training protocols are applied; FedGKT (He et al., 2020a) further demonstrates an advanced algorithm that requires exchanging activation maps and distilled logits (the hidden vector that feeds into the softmax function for probabilistic prediction), beyond gradients/weights. FedML offers the flexibility of these implementations by wrapping PyTorch RPC API as a design pattern called "worker-oriented programming interface" (He et al., 2020b). Specifically, developers no longer need to maintain communications and distributed contexts and only focus on message definition for algorithms but enjoy the underlying communication optimization introduced in Section 3. From the comparison of the gRPC and PyTorch RPC implementation in FedML, we see that the three use cases can be supported by the same communication backend with a unified but flexible message definition: PyTorch RPC can automatically take care of FedML `Message` class to distinguish the communication between the tensor and other protocol messages, but the gRPC implementation has to handle PB protocol, which requires many more LoC.

5 Related Work

Distributed training solutions can be categorized into data parallel, model parallel, and hybrid parallel paradigms. Horovod (Sergeev & Balso, 2018) is a typical data parallel solution. Model parallelism partitions the model and places different shards on different processes, which helps to scale large models to multiple GPUs and machines. GPipe (Huang et al., 2019) partitions a sequence of layers at operator boundaries, and feeds micro-batches into those layers as a pipeline. PipeDream (Narayanan et al., 2019) and PipeMare (Yang et al., 2021a) speed up pipeline parallelism by removing synchronization barriers at the cost of allowing gradient staleness. Hybrid parallelism combines data and model parallel solutions. PipeTransformer (He et al., 2021) employs synchronous pipeline parallel within one machine and data parallel across machines. It gradually freezes the stack of layers in order to reduce the number of active parameters and spawns more processes on freed resources to increase data parallel width. Besides partitioning at operator boundaries, Megatron-LM (Shoeybi et al., 2019), Mesh-TensorFlow (Shazeer et al., 2018), GShard (Chen et al., 2021), and ToFu (Wang et al., 2019) explored sharding individual layers across processes when applicable, which usually requires either custom implementation for sharded layers or compiler support to decompose layers into unit operators and insert communications for activations and gradients. ZeRO (Rajbhandari et al., 2019) circumvents the requirements by sharding only model parameters instead of computation, where every process fetches parameters of a layer from its owner before computation and discards the layer afterward. These technologies push the limit of distributed training for different domains in different directions but impose constraints on aspects like model structure and training paradigm. Moreover, although they share many common components, such as tensor communication, remote data manipulation, and distributed autograd, those components are implemented independently, differently, and repeatedly. The next distributed training innovation will likely still need to start from scratch.

Despite the limitations, many existing training tasks in natural language processing and computer vision can fit very well into the aforementioned paradigms. However, no one can anticipate what requirements might emerge in the future in these two domains. Moreover, other domains such as federated learning, reinforcement learning, and graph learning already suffer from the lack of flexibility. As a result, developers in such domains have started to construct their own distributed training solutions. FedML (He et al., 2020b) supports generic federated learning typologies which cannot comply with existing data and model parallel frameworks. Therefore, it creates its own message-passing protocol on top of gRPC to power higher-level features. TorchBeast (Küttler et al., 2019) is a reinforcement learning library that aims at training agents in both local and distributed environments, which also relies on gRPC for communications. BigGraph (Lerer et al., 2019) helps to learn graph embeddings for large graphs with up to billions of entities and trillions of edges. It wraps PyTorch `send` and `recv` APIs into its own RPC services. General-purpose distributed computing frameworks, such as Ray (Ray, 2023; Moritz et al., 2018) and Dask (Dask, 2023), can help to set up distributed applications but still lacks supports for efficient tensor communication and distributed autograd. Evaluations presented in Section 4 show that building training applications using universal RPC frameworks can be slow and verbose.

6 Conclusion

In this paper, we introduce PyTorch RPC, a generic solution for distributed deep learning. The RPC package offers greater flexibility compared to specialized distributed training tools and superior simplicity and efficiency compared to generalized distributed computing frameworks. Evaluations show that PyTorch RPC outperforms gRPC by up to two orders of magnitude in tensor communication. Case studies confirm that it helps to simplify the implementation of reinforcement learning, large language model, recommendation model, and federated learning applications.

REFERENCES

Biswas, R., Lu, X., and Panda, D. K. Accelerating tensorflow with adaptive rdma-based grpc. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pp. 2–11. IEEE, 2018.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), Advances in Neural Information Processing Systems, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Chen, D., Lepikhin, D. D., Lee, H., Krikun, M., Shazeer, N., Firat, O., Huang, Y., Xu, Y., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. In Tenth International Conference on Learning Representations, 2021.

Dask. Dask: Scalable analytics in Python. https://dask.org/, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv preprint arXiv:2101.03961, 2021.

gRPC. gRPC: A high performance, open source universal RPC framework. https://grpc.io/, 2021.

Gupta, U., Wang, X., Naumov, M., Wu, C., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K. M., Jia, B., Lee, H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., and Zhang, X. The architectural implications of facebook's dnn-based personalized recommendation. CoRR, abs/1906.03109, 2019. URL https://arxiv.org/abs/1906.03109.

Gym. OpenAI Gym. https://gym.openai.com/, 2021.

Hardy, S., Henecka, W., Ivey-Law, H., Nock, R., Patrini, G., Smith, G., and Thorne, B. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. arXiv preprint arXiv:1711.10677, 2017.

He, C., Annavaram, M., and Avestimehr, S. Group knowledge transfer: Federated learning of large cnns at the edge. arXiv: Learning, 2020a.

He, C., Li, S., So, J., Zeng, X., Zhang, M., Wang, H., Wang, X., Vepakomma, P., Singh, A., Qiu, H., et al. FedML: A Research Library and Benchmark for Federated Machine Learning. In NeurIPS Workshop on Scalability, Privacy, and Security in Federated Learning, 2020b.

He, C., Li, S., Soltanolkotabi, M., and Avestimehr, S. Pipetransformer: Automated elastic pipelining for distributed training of transformers. In Thirty-eighth International Conference on Machine Learning, 2021.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Advances in Neural Information Processing Systems, pp. 103–112, 2019.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. Proceedings of Machine Learning and Systems, 2019.

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A., Bonawitz, K., Charles, Z. B., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Rouayheb, S., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konecný, J., Korolova, A., Koushanfar, F., Koyejo, O., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F., Yu, H., and Zhao, S. Advances and open problems in federated learning. Found. Trends Mach. Learn., 14:1–210, 2021.

Kim, C., Lee, H., Jeong, M., Baek, W., Yoon, B., Kim, I., Lim, S., and Kim, S. torchgpipe: On-the-fly pipeline parallelism for training giant models. arXiv preprint arXiv:2004.09910, 2020.

Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. Federated learning: Strategies for improving communication efficiency. arXiv preprint arXiv:1610.05492, 2016.

Küttler, H., Nardelli, N., Lavril, T., Selvatici, M., Sivakumar, V., Rocktäschel, T., and Grefenstette, E. Torchbeast: A pytorch platform for distributed rl. arXiv preprint arXiv:1910.03552, 2019.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. PyTorch-BigGraph: A Large-scale Graph Embedding System. In Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA, 2019.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pp. 583–598, 2014.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch Distributed: Experiences on accelerating data parallel training. In VLDB, 2020.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. arXiv preprint arXiv:1609.07843, 2016. under the Creative Commons Attribution-ShareAlike License.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning, 2016.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging {AI} applications. In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), pp. 561–577, 2018.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 1–15, 2019.

Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems. CoRR, abs/1906.00091, 2019. URL https://arxiv.org/abs/1906.00091.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimization towards training a trillion parameter models. arXiv preprint arXiv:1910.02054, 2019.

Ray. Ray: Fast and Simple Distributed Computing. https://ray.io/, 2023.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., and Weinberger, K. Q. (eds.), Advances in Neural Information Processing Systems, volume 24, pp. 693–701. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799, 2018.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. In Advances in Neural Information Processing Systems, pp. 10414–10423, 2018.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

Thangakrishnan, I., Cavdar, D., Karakus, C., Ghai, P., Selivonchyk, Y., and Pruce, C. Herring: rethinking the parameter server at scale for the cloud. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, pp. 1–13, 2020.

Wang, J., Charles, Z. B., Xu, Z., Joshi, G., McMahan, H. B., Arcas, B. A. Y., Al-Shedivat, M., Andrew, G., Avestimehr, S., Daly, K., Data, D., Diggavi, S., Eichner, H., Gadhikar, A., Garrett, Z., Girgis, A. M., Hanzely, F., Hard, A., He, C., Horvath, S., Huo, Z., Ingerman, A., Jaggi, M., Javidi, T., Kairouz, P., Kale, S., Karimireddy, S. P. R., Konecný, J., Koyejo, S., Li, T., Liu, L., Mohri, M., Qi, H., Reddi, S. J., Richtárik, P., Singhal, K., Smith, V., Soltanolkotabi, M., Song, W., Suresh, A. T., Stich, S. U., Talwalkar, A. S., Wang, H., Woodworth, B. E., Wu, S., Yu, F. X., Yuan, H., Zaheer, M., Zhang, M., Zhang, T., Zheng, C., Zhu, C., and Zhu, W. A field guide to federated optimization. ArXiv, abs/2107.06917, 2021.

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In Proceedings of the Fourteenth EuroSys Conference 2019, pp. 1–17, 2019.

Xue, J., Miao, Y., Chen, C., Wu, M., Zhang, L., and Zhou, L. Fast distributed deep learning over rdma. In Proceedings of the Fourteenth EuroSys Conference 2019, pp. 1–14, 2019.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C., and De Sa, C. Pipemare: Asynchronous pipeline parallel dnn training. In Proceedings of Machine Learning and Systems, volume 3, 2021a.

Yang, C.-S., So, J., He, C., Li, S., Yu, Q., and Avestimehr, S. Lightsecagg: Rethinking secure aggregation in federated learning. arXiv preprint arXiv:2109.14236, 2021b.

Yang, Q., Liu, Y., Chen, T., and Tong, Y. Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol., 10(2), January 2019. ISSN 2157-6904. doi: 10.1145/3298981. URL https://doi.org/10.1145/3298981.

Zha, D., Feng, L., Luo, L., Bhushanam, B., Liu, Z., Hu, Y., Nie, J., Huang, Y., Tian, Y., Kejariwal, A., and Hu, X. Pre-trained neural cost models for efficient embedding table sharding in deep learning recommendation models. In Proceedings of Machine Learning and Systems, volume 6, 2023.

Zhang, B., Luo, L., Liu, X., Li, J., Chen, Z., Zhang, W., Wei, X., Hao, Y., Tsang, M., Wang, W., et al. Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction. arXiv preprint arXiv:2203.11014, 2022.